Appendix C
Memory Hierarchy

Levels of the Memory Hierarchy

Adapted from Patterson and Hennessy
(Morgan Kaufman Pubs)
Memory Hierarchy: Apple iMac G5

<table>
<thead>
<tr>
<th></th>
<th>Reg</th>
<th>L1 Inst</th>
<th>L1 Data</th>
<th>L2</th>
<th>DRAM</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>1K</td>
<td>64K</td>
<td>32K</td>
<td>512K</td>
<td>256M</td>
<td>80G</td>
</tr>
<tr>
<td>Latency</td>
<td>Cycles, Time</td>
<td>1, 0.6 ns</td>
<td>3, 1.9 ns</td>
<td>3, 1.9 ns</td>
<td>11, 6.9 ns</td>
<td>88, 55 ns</td>
</tr>
</tbody>
</table>

iMac G5
1.6 GHz

Goal: Illusion of large, fast, cheap memory

Let programs address a memory space that scales to the disk size, at a speed that is usually as fast as register access

Adapted from Patterson and Hennessey
(Morgan Kaufman Pub)
The Principle of Locality

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.

- Two Different Types of Locality:
  - **Temporal Locality** (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - **Spatial Locality** (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)

- Last 15 years, HW relied on locality for speed

It is a property of programs which is exploited in machine design.

Memory Hierarchy: Terminology

- **Hit**: data appears in some block in the upper level (example: Block X)
  - **Hit Rate**: the fraction of memory access found in the upper level
  - **Hit Time**: Time to access the upper level which consists of RAM access time + Time to determine hit/miss

- **Miss**: data needs to be retrieved from a block in the lower level (Block Y)
  - **Miss Rate** = 1 - (Hit Rate)
  - **Miss Penalty**: Time to replace a block in the upper level + Time to deliver the block the processor

- Hit Time << Miss Penalty
Cache Measures

- **Hit rate**: fraction found in that level
  - So high that usually talk about **Miss rate**
  - Miss rate fallacy: as MIPS to CPU performance, miss rate to average memory access time in memory
- **Average memory-access time**
  \[ \text{Avg. memory-access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty} \]
  (ns or clocks)
- **Miss penalty**: time to replace a block from lower level, including time to replace in CPU
  - **access time**: time to lower level
    \[ = f(\text{latency to lower level}) \]
  - **transfer time**: time to transfer block
    \[ = f(\text{BW between upper & lower levels}) \]

4 Questions for Memory Hierarchy

- Q1: Where can a block be placed in the upper level?  
  *(Block placement)*
- Q2: How is a block found if it is in the upper level?  
  *(Block identification)*
- Q3: Which block should be replaced on a miss?  
  *(Block replacement)*
- Q4: What happens on a write?  
  *(Write strategy)*
Q1: Where can a block be placed in the upper level?

- Block 12 placed in 8 block cache:
  - Fully associative, direct mapped, 2-way set associative
  - S.A. Mapping = Block Number Modulo Number Sets

<table>
<thead>
<tr>
<th>Cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1234567</td>
</tr>
<tr>
<td>1</td>
<td>1234567</td>
</tr>
<tr>
<td>2</td>
<td>1234567</td>
</tr>
<tr>
<td>3</td>
<td>1234567</td>
</tr>
<tr>
<td>4</td>
<td>1234567</td>
</tr>
<tr>
<td>5</td>
<td>1234567</td>
</tr>
<tr>
<td>6</td>
<td>1234567</td>
</tr>
<tr>
<td>7</td>
<td>1234567</td>
</tr>
<tr>
<td>8</td>
<td>1234567</td>
</tr>
<tr>
<td>9</td>
<td>1234567</td>
</tr>
<tr>
<td>10</td>
<td>1234567</td>
</tr>
<tr>
<td>11</td>
<td>1234567</td>
</tr>
<tr>
<td>12</td>
<td>1234567</td>
</tr>
<tr>
<td>13</td>
<td>1234567</td>
</tr>
<tr>
<td>14</td>
<td>1234567</td>
</tr>
<tr>
<td>15</td>
<td>1234567</td>
</tr>
</tbody>
</table>

Q2: How is a block found if it is in the upper level?

- Tag on each block
  - No need to check index or block offset
- Increasing associativity shrinks index, expands tag
Q3: After a cache read miss, if there are no empty cache blocks, which block should be removed from the cache?

**The Least Recently Used (LRU) block?** Appealing, but hard to implement for high associativity

**A randomly chosen block?** Easy to implement, how well does it work?

**Miss Rate for 2-way Set Associative Cache**

<table>
<thead>
<tr>
<th>Size</th>
<th>Random</th>
<th>LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 KB</td>
<td>5.7%</td>
<td>5.2%</td>
</tr>
<tr>
<td>64 KB</td>
<td>2.0%</td>
<td>1.9%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.17%</td>
<td>1.15%</td>
</tr>
</tbody>
</table>

Also, try other LRU approx.

Adapted from Patterson and Hennessy (Morgan Kaufman Pubs)

Q4: What happens on a write?

<table>
<thead>
<tr>
<th>Policy</th>
<th>Write-Through</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data written to cache block</td>
<td>Write data only to the cache</td>
</tr>
<tr>
<td></td>
<td>also written to lower-level memory</td>
<td>Update lower level when a block falls out of the cache</td>
</tr>
<tr>
<td>Debug</td>
<td>Easy</td>
<td>Hard</td>
</tr>
<tr>
<td>Do read misses produce writes?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Do repeated writes make it to lower level?</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Additional option -- let writes to an un-cached address allocate a new cache line (“write-allocate”).

Adapted from Patterson and Hennessy (Morgan Kaufman Pubs)
Write Buffers for Write-Through Caches

Holds data awaiting write-through to lower level memory

Q. Why a write buffer?  
A. So CPU doesn’t stall

Q. Why a buffer, why not just one register?  
A. Bursts of writes are common.

5 Basic Cache Optimizations

- Reducing Miss Rate
  1. Larger Block size (compulsory misses)
  2. Larger Cache size (capacity misses)
  3. Higher Associativity (conflict misses)

- Reducing Miss Penalty
  4. Multilevel Caches

- Reducing hit time
  5. Giving Reads Priority over Writes
    - E.g., Read complete before earlier writes in write buffer

Adapted from Patterson and Hennessey  
(Morgan Kaufmann Pubs)
The Limits of Physical Addressing

“Physical addresses” of memory locations

All programs share one address space:
The physical address space

Machine language programs must be aware of the machine organization

No way to prevent a program from accessing any machine resource
Solution: Add a Layer of Indirection

User programs run in an standardized virtual address space

Address Translation hardware managed by the operating system (OS) maps virtual address to physical memory

Three Advantages of Virtual Memory

- **Translation:**
  - Program can be given consistent view of memory, even though physical memory is scrambled
  - Makes multithreading reasonable (now used a lot!)
  - Only the most important part of program ("Working Set") must be in physical memory.
  - Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later.

- **Protection:**
  - Different threads (or processes) protected from each other.
  - Different pages can be given special behavior
    - (Read Only, Invisible to user programs, etc).
  - Kernel data protected from User programs
  - Very important for protection from malicious programs

- **Sharing:**
  - Can map same physical page to multiple users ("Shared memory")
Details of Page Table

- Page table maps virtual page numbers to physical frames ("PTE" = Page Table Entry)
- Virtual memory => treat memory ≈ cache for disk

Adapted from Patterson and Hennessey (Morgan Kaufman Pubs)

MIPS Address Translation: How does it work?

"Virtual Addresses"

CPU

D0-D31

Translation Look-Aside Buffer (TLB)

Translation Look-Aside Buffer (TLB)
A small fully-associative cache of mappings from virtual to physical addresses
TLB also contains protection bits for virtual address

Adapted from Patterson and Hennessey (Morgan Kaufman Pubs)
Use virtual addresses for cache?

“Virtual Addresses”

CPU

A0-A31

D0-D31

Virtual Cache

Virtual

Physical

Translation

Look-Aside

Buffer

(TLB)

“A0-A31

Main Memory

D0-D31

“Physical Addresses”

Only use TLB on a cache miss!

Downside: a subtle, fatal problem. What is it?

A. Synonym problem. If two address spaces share a physical frame, data may be in cache twice. Maintaining consistency is a nightmare.

Cache choices versus VM choices

- **Caches**
  - The most important thing is speed
  - We choose:
    » Direct-Mapped or small Set Associative
    » Fast selection of block to be replaced (e.g., Random)
    » Write-through

- **Virtual memory**
  - The most important thing is to minimize misses
  - We choose:
    » Full Associative or large Set Associative
    » The best (practical) algorithm for replacing blocks (e.g., LRU)
    » Write-back

Adapted from Patterson and Hennessy
(Morgan Kaufman Pubs)